



Attorney Docket No.
095313.00001

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent of:

Marc D. Van Heyningen

Examiner: L. Son

U.S. Pat. App. No.: 09/782,593

Group Art Unit: 2135

Filing Date: February 12, 2001

For: METHOD AND APPARATUS FOR PROVIDING SECURE STREAMING DATA
TRANSMISSION FACILITIES USING UNRELIABLE PROTOCOLS

DECLARATION UNDER 37 C.F.R. § 1.131

Commissioner for Patents
P.O. Box 1450,
Alexandria, Virginia 22313-1450

RECEIVED

NOV 24 2004

Sir:

Technology Center 2100

I, Marc D. Van Heyningen, do hereby declare as follows:

1. I am named as the inventor in U.S. Patent Application No. 09/782,593, filed February 12, 2001. I am an employee of Aventail Corporation having a place of business at 808 Howell Street, Second Floor, Seattle, Washington 98101, and I am over the age of eighteen years.

2. I am advised that Exhibit A contains a true and correct copy of the claims currently pending in U.S. Patent Application No. 09/782,593, namely claims 1-22. I have read and understand these claims.

3. I also have been informed that the United States Patent and Trademark Office has rejected claims 4-7 and 10-22 as contained in Exhibit A based upon U.S. Patent Application No. 2002/0094085 A1 to Roberts. I understand that the Roberts patent claims an effective U.S. filing

U.S. Pat. App. No.: 09/782,593
Atty. Docket No.: 005313.00001

date of January 16, 2001.

4. I have reviewed (a) the computer code file entitled "sslctx.c" (a printed copy of which is attached as Exhibit B), (b) the computer code file entitled "sslrec.c" (a printed copy of which is attached as Exhibit C), (c) the computer code file entitled "ciphers.c" (a printed copy of which is attached as Exhibit D), and (d) the computer code entitled "sslenv.c" (a printed copy of which is attached as Exhibit E). I created each of these files of computer code by revising existing computer code to implement the invention as described in my patent application and recited in claims 4-7 and 10-22 (Exhibit A). I have obtained each of these files of computer code from our company's records. The copy of each of these files of computer code has been changed to include line numbers for the code lines. The line numbers were added in Exhibit B through Exhibit E to make the discussion below easier to follow, i.e., so that the cited code line numbers below would be readily available with the attached copy of the code.

5. As will be described in more detail below, the attached computer code, when implemented on a computer system, allows one to encrypt and transmit data records between a first computer and a second computer using an unreliable communication protocol in the manner described in my patent application and recited in claims 4-7 and 10-22 (Exhibit A).

6. One feature of a claim in my patent application relates to encrypting and transmitting data records between a first computer and a second computer using an unreliable communication protocol wherein each data record is encrypted by incorporating a nonce and without reference to a previously transmitted data record (note, for example, claims 4-6).

7. Referencing the source code, portions of the code that implement functionality relating

U.S. Pat. App. No.: 09/782,593
Atty. Docket No.: 005313.00001

to my invention typically reference the term "SSLoppy", the working name for the development of the invention when this code was created. This term refers to the features provided by the invention that allows Secure Sockets Layer (SSL) records to be dropped and reordered during communication without disruption, which thus offer a "sloppy" variation of the SSL communication technique. The role of the first computer is included in: `sslrec.c`, lines 326-336, where the computer code adds extra room to a record for the inclusion of an explicit nonce, lines 354-378, where the computer code generates the nonce randomly, uses it in place of the sequence number in computing the MAC, and includes it in the record being built, lines 406-410, where the computer code passes this nonce to the encryption function, and in file `cipher.c`, lines 252-257, where the computer code uses the passed nonce value as an initial vector (IV), if present. The role of the second computer in reading these records is described in: file `sslrec.c`, lines 450-455, where the code parses the nonce out of incoming records, lines 466-475, where the code passes this nonce to the decryption function, and in file `cipher.c`, lines 323-328, where the code uses the passed nonce as the initial vector (IV), if present.

8. Another feature of a claim in my patent application relates to encrypting and transmitting data records between a first computer and a second computer using an unreliable communication protocol wherein an indicator is embedded in each of the data records indicating that the data records are encrypted according to an encryption scheme that encrypts records without regard to any previously transmitted data records, and the second computer determines whether the indicator is present in each record and, in response to determining that the indicator is not present, processes each such record differently than if the indicator is set (note, for example, claim 7).

U.S. Pat. App. No.: 09/782,593
Atty. Docket No.: 005313.00001

9. Referencing the source code, in file sslrec.c, the role of the first computer in sending its different form of data is included in: lines 326-336, where the software code allocates extra space in the record for the nonce, if necessary, lines 354-391, where the ssloppy flag is used to decide whether to generate the record differently, lines 406-410, where the same flag is used to decide whether to use the nonce as the initial vector (IV) for the symmetric cipher algorithm, and lines 424-428, where the flag is used to decide whether to increment the sequence number. The role of the second computer, in receiving a packet of a different form of data, is included in: lines 95-99, where the software code checks a flag in the incoming record to verify that it is only receiving the new type of reorderable data when expected, lines 221-222, where the software code uses the same flag to determine whether to increment the sequence number, lines 450-454, where the software code uses the same flag to determine whether to extract the nonce from the record, lines 466-480, where the software code uses the same flag to decide whether to use this nonce as the initial vector (IV) when decrypting the record, and lines 497-517, where the software code uses the same flag to decide whether to use the nonce in place of the sequence number when verifying the MAC.

10. Still another feature of a claim in my patent application relates to securely transmitting a plurality of data records between a client computer and a proxy server using an unreliable communication protocol, that include the steps of establishing a reliable connection between the client computer and the proxy server, exchanging encryption credentials between the client computer and the proxy server over the reliable connection generating a nonce for each of a plurality of data records, wherein each nonce comprises an initialization vector necessary to

U.S. Pat. App. No.: 09/782,593
Atty. Docket No.: 005313.00001

decrypt a corresponding one of the plurality of data records, using the nonce to encrypt each of the plurality of data records and appending the nonce to each of the plurality of data records, transmitting the plurality of encrypted data records from the client computer to the proxy server using an unreliable communication protocol, and, in the proxy server, decrypting each of the plurality of encrypted data records using a corresponding nonce extracted from each data record and a previously shared encryption key (note claims 10-15).

11. The proxy server and client architecture described above was already a commercial shipping product from Aventail, Inc. by January 16, 2001. Prior to the enhancements offered by the implementation of the invention, the product utilized the unreliable communication protocol "User Data Protocol" (UDP) to relay datagram records over a communication medium without any cryptographic protection. Comments in the software code refer to this process as "UDP Naked." The addition of the invention replaced "UDP Naked" with the "SSLoppy" process (as described above) thereby allowing communications to be cryptographically protected. Referencing the file sslenv.c, lines 1056-1067, for transmitting a datagram with this SSLoppy feature enabled, the SSLoppy encryption feature is turned on for the processing of a record by calling SSLSetSloppyMode to 1, and, subsequent to processing of the record, resetting this value back to 0 (described in lines 1131, 1145, 1153, 1183, 1220, 1244, 1252, 1259, 1278, 1286, 1314, 1347, 1359, and 1397). This software code processes only designated datagrams using the SSLoppy encryption process, and provides for standard SSL processing of data records being exchanged via a reliable communication protocol. The underlying functionality for selecting the SSLoppy encryption process is in sslctx.c, lines 899-911, where this function call sets the ctx-

U.S. Pat. App. No.: 09/782,593
Atty. Docket No.: 005313.00001

>ssloppy flag. This flag is used to read and write records in the mode documented in paragraph 9 above.

12. Still another feature of a claim in my patent application relates to a system for securely transmitting data using an unreliable protocol that includes a first computer comprising a communication protocol client function operable in conjunction with an application program to transmit data records securely using an unreliable protocol, and a second computer coupled to the first computer and comprising a communication protocol server function operable in conjunction with the communication protocol client function to receive data records securely using the unreliable communication protocol, wherein the communication protocol client function encrypts each data record using a nonce and an encryption key and appends the respective nonce to each of the encrypted data records, and wherein the communication protocol server function decrypts each of the data records using the respectively appended nonce and the encryption key (note claims 16-22).

13. Supporting code is the same as that from paragraph 11 above.

14. The "source code" software code existed in the form shown in Exhibits B through E prior to January 16, 2001. This computer code received unit testing and was used as a proof-of-concept in the development process. Using this software code, the invention performed in its intended manner prior to January 16, 2001, as described in our patent application and claims 4-7 and 10-22 (as shown in Exhibit A), and thus was actually reduced to practice prior to January 16, 2001.

U.S. Pat. App. No.: 09/782,593
Atty. Docket No.: 005313.00001

DECLARATION IN LIEU OF OATH

15. I further declare that all information stated herein based upon my own knowledge is true and that all information stated herein based on information and belief is believed to be true, and further that the statements made in this Declaration were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of this application or any patent issuing based on this application.

Date:

By: 
Marc Van Heyningen, Inventor

EXHIBIT A

1. A method of transmitting data securely over a computer network, comprising the steps of:

- (1) establishing a communication path between a first computer and a second computer;
- (2) encrypting and transmitting data records between the first computer and the second computer using an unreliable communication protocol, wherein each data record is encrypted without reference to a previously transmitted data record; and
- (3) in the second computer, receiving and decrypting the data records transmitted in step (2) without reference to a previously received data record.

2. The method of claim 1, further comprising the step of, prior to step (1), establishing a reliable communication path between the first computer and the second computer and exchanging security credentials over the reliable communication path.

3. The method of claim 2, wherein the step of exchanging security credentials comprises the step of exchanging an encryption key that is used to encrypt the data records in step (2).

4. The method of claim 1, wherein step (2) comprises the step of incorporating a nonce in each data record that is used by the second computer in combination with a previously shared encryption key to decrypt each of the data records in step (3).

5. The method of claim 4, wherein the nonce comprises a random number.

6. The method of claim 4, further comprising the step of, in the second computer, verifying that the nonce has not previously been received in a previously transmitted data record.

7. The method of claim 1, wherein step (2) comprises the step of embedding an indicator in each of the data records indicating that the data records are encrypted according to an encryption scheme that encrypts records without regard to any previously transmitted data records, and

wherein step (3) comprises the step of determining whether the indicator is present in each record and, in response to determining that the indicator is not present, processing each such record differently than if the indicator is set.

8. The method of claim 1, wherein step (1) is performed using the Transmission Control Protocol, and wherein step (2) is performed using the User Datagram Protocol.

9. The method of claim 1, wherein step (2) is performed by a proxy server that encrypts data records received from another server.

10. A method of securely transmitting a plurality of data records between a client computer and a proxy server using an unreliable communication protocol, comprising the steps of:

- (1) establishing a reliable connection between the client computer and the proxy server;
- (2) exchanging encryption credentials between the client computer and the proxy server over the reliable connection;

(3) generating a nonce for each of a plurality of data records, wherein each nonce comprises an initialization vector necessary to decrypt a corresponding one of the plurality of data records;

(4) using the nonce to encrypt each of the plurality of data records and appending the nonce to each of the plurality of data records;

(5) transmitting the plurality of data records encrypted in step (4) from the client computer to the proxy server using an unreliable communication protocol; and

(6) in the proxy server, decrypting each of the plurality of encrypted data records using a corresponding nonce extracted from each data record and a previously shared encryption key.

11. The method of claim 10, wherein step (6) comprises the step of checking to determine whether each data record received from the client computer is formatted according to a secure unreliable transmission format and, if a particular record is not formatted according to a secure unreliable transmission format, bypassing the decryption using the corresponding nonce.

12. The method of claim 10, wherein step (3) comprises the step of generating a random number as each nonce.

13. The method of claim 10, wherein step (1) is performed using Transmission Control Protocol, and wherein step (5) is performed using User Datagram Protocol.

14. The method of claim 10, wherein step (6) is performed using an encryption key previously shared using a reliable communication protocol.

15. The method of claim 14, wherein the reliable communication protocol is Transmission Control Protocol.

16. A system for securely transmitting data using an unreliable protocol, comprising:
a first computer comprising a communication protocol client function operable in conjunction with an application program to transmit data records securely using an unreliable protocol; and

a second computer coupled to the first computer and comprising a communication protocol server function operable in conjunction with the communication protocol client function to receive data records securely using the unreliable communication protocol,

wherein the communication protocol client function encrypts each data record using a nonce and an encryption key and appends the respective nonce to each of the encrypted data records; and

wherein the communication protocol server function decrypts each of the data records using the respectively appended nonce and the encryption key.

17. The system of claim 16, wherein the communication protocol client function exchanges encryption credentials with the communication protocol server function using a reliable communication protocol.

18. The system of claim 17, wherein the unreliable communication protocol comprises the User Datagram Protocol, and wherein the reliable communication protocol comprises the Transmission Control Protocol.

19. The system of claim 16, wherein the communication protocol client function and the communication protocol server function are compatible with the SOCKS communication protocol.

20. The system of claim 16, wherein the communication protocol client function and the communication protocol server function are compatible with the SSL/TLS communication protocol.

21. The system of claim 16, wherein the second computer comprises a proxy server that forwards decrypted records received from the first computer to a server computer.

22. The system of claim 16, wherein the second computer comprises a record detector that determines whether an indicator has been set in each data record received from the first computer and, if the indicator has not been set, bypassing decryption in the server computer.

EXHIBIT B

```
0 #include <stdio.h>
1
2 /* #define HYPER_DEBUG 1 */
3
4 /* *****
5  File: sslctx.c
6
7  SSL Plus: Security Integration Suite(tm)
8  Version 1.1.1 -- August 11, 1997
9
10 Copyright (c)1996, 1997 by Consensus Development Corporation
11     Copyright (c)1997, 1998 by Aventail Corporation
12
13 Portions of this software are based on SSLRef(tm) 3.0, which is
14 Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
15 was developed by Netscape Communications Corporation and Consensus
16 Development Corporation.
17
18 In order to obtain this software, your company must have signed
19 either a PRODUCT EVALUATION LICENSE (a copy of which is included in
20 the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
21 licenses have different limitations regarding how you are allowed to
22 use the software. Before retrieving (or using) this software, you
23 *must* ascertain which of these licenses your company currently
24 holds. Then, by retrieving (or using) this software you agree to
25 abide by the particular terms of that license. If you do not agree
26 to abide by the particular terms of that license, than you must
27 immediately delete this software. If your company does not have a
28 signed license of either kind, then you must either contact
29 Consensus Development and execute a valid license before retrieving
30 (or using) this software, or immediately delete this software.
31
32 *****
33
34 File: sslctx.c      SSLContext accessors
35
36 Functions called by the end user which configure an SSLContext
37 structure or access data stored there.
38
39 <because of the size of this file and the fact that most of it is not relevant, we include only the relevant function SSLSetSloppyMode here>
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89 SSLerr CDECL SSLSetSloppyMode(SSLContext *ctx, int m)
90 {
91     if(ctx == NULL)
92         return SSLUnknownErr;
93
94     if((ctx->selectedCipherSpec->cipher->blockSize == 0) &&
95        (ctx->selectedCipherSpec->cipher->keySize > 0))
96         return SSLProtocolErr;
97
98     ctx->ssloppy = m;
99
100    return SSLNoErr;
101 }
```

EXHIBIT C

```
0 /* *****
1   File: sslrec.c
2
3   SSL Plus: Security Integration Suite(tm)
4   Version 1.1.1 -- August 11, 1997
5
6   Copyright (c)1996, 1997 by Consensus Development Corporation
7       Copyright (c)1997, 1998 by Aventail Corporation
8
9   Portions of this software are based on SSLRef(tm) 3.0, which is
10  Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
11  was developed by Netscape Communications Corporation and Consensus
12  Development Corporation.
13
14  In order to obtain this software, your company must have signed
15  either a PRODUCT EVALUATION LICENSE (a copy of which is included in
16  the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
17  licenses have different limitations regarding how you are allowed to
18  use the software. Before retrieving (or using) this software, you
19  *must* ascertain which of these licenses your company currently
20  holds. Then, by retrieving (or using) this software you agree to
21  abide by the particular terms of that license. If you do not agree
22  to abide by the particular terms of that license, than you must
23  immediately delete this software. If your company does not have a
24  signed license of either kind, then you must either contact
25  Consensus Development and execute a valid license before retrieving
26  (or using) this software, or immediately delete this software.
27
28  *****
29
30  File: sslrec.c      Encryption, decryption and MACing of data
31
32  All the transformations which occur between plaintext and the
33  secured, authenticated data that goes out over the wire. Also,
34  detects incoming SSL 2 hello messages and hands them off to the SSL 2
35  record layer (and hands all SSL 2 reading & writing off to the SSL 2
36  layer).
37
38  ***** */
39
40 /* #define HYPER_DEBUG 1 */
41
42 #ifdef HYPER_DEBUG
43 #include <stdio.h>
44 #endif
45
46 #ifndef _SSL_H_
47 #include "ssl.h"
48 #endif
49
50 #ifndef _SSLREC_H_
51 #include "sslrec.h"
52 #endif
53
54 #ifndef _SSLALLOC_H_
55 #include "sslalloc.h"
56 #endif
57
58 #ifndef _CRYPTYPE_H_
59 #include "cryptype.h"
60 #endif
61
62 #ifndef _SSLCTX_H_
63 #include "sslctx.h"
64 #endif
65
66 #ifndef _SSLALERT_H_
67 #include "sslalert.h"
68 #endif
69
70 #ifndef _SSL2_H_
```

```
71 #include "ssl2.h"
72 #endif
73
74 #include <string.h>
75
76 static SSLerr DecryptSSLRecord(uint8 type, SSLBuffer *payload, SSLContext *ctx);
77 static SSLerr VerifyMAC(uint8 type, SSLBuffer data, uint8 *compareMAC, uint64 seqNo, SSLContext
    *ctx);
78 static SSLerr ComputeMAC(uint8 type, SSLBuffer data, SSLBuffer mac, uint64 seqNo, SSLBuffer
    secret, CipherContext *cipherCtx, SSLContext *ctx);
79 static uint8* SSLEncodeUInt64(uint8 *p, uint64 value);
80
81 /* ReadSSLRecord
82  * Attempt to read & decrypt an SSL record.
83  */
84 SSLerr
85 SSLReadRecord(SSLRecord *rec, SSLContext *ctx)
86 {
87     SSLerr      err;
88     uint32      len, contentLen;
89     uint8       *progress;
90     SSLBuffer    readData, cipherFragment;
91
92 #ifdef HYPER_DEBUG
93     fprintf(stderr, "Got into SSLReadRecord, whee!\n");
94 #endif
95
96     /* if we get UDP data when we aren't expecting it, that's really bad,
97      so report an appropriate error. */
98     if((rec->contentType == SSL_application_data_ssloppy) &&
99         (! ctx->ssloppy))
100         return SSLProtocolErr;
101
102     if (!ctx->partialReadBuffer.data || ctx->partialReadBuffer.length < 5)
103     {
104         if (ctx->partialReadBuffer.data)
105             if ((err = SSLFreeBuffer(&ctx->partialReadBuffer, &ctx->sysCtx)) != 0)
106             {
107                 SSLFatalSessionAlert(alert_close_notify, ctx);
108                 return ERR(err);
109             }
110         if ((err = SSLAllocBuffer(&ctx->partialReadBuffer, DEFAULT_BUFFER_SIZE, &ctx->sysCtx))
111             != 0)
112         {
113             SSLFatalSessionAlert(alert_close_notify, ctx);
114             return ERR(err);
115         }
116     }
117
118     if (ctx->protocolVersion == SSL_Version_Undetermined ||
119         ctx->protocolVersion == SSL_Version_3_0_With_2_0_Hello)
120     {
121         if (ctx->amountRead < 1)
122         {
123             readData.length = 1 - ctx->amountRead;
124             readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
125             len = readData.length;
126             if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
127             {
128                 if (err == SSLWouldBlockErr)
129                     ctx->amountRead += len;
130                 else
131                     SSLFatalSessionAlert(alert_close_notify, ctx);
132                 return err;
133             }
134             ctx->amountRead += len;
135         }
136     }
137
138     /* In undetermined cases, if the first byte isn't in the range of SSL 3.0
139     * record types, this is an SSL 2.0 record
140     */
141     switch (ctx->protocolVersion)
142     {
143     case SSL_Version_Undetermined:
144     case SSL_Version_3_0_With_2_0_Hello:
145         if (ctx->partialReadBuffer.data[0] < SSL_smallest_3_0_type ||
146             ctx->partialReadBuffer.data[0] > SSL_largest_3_0_type)
```



```
139         return SSL2ReadRecord(rec, ctx);
140     else
141         break;
142     case SSL_Version_2_0:
143         return SSL2ReadRecord(rec, ctx);
144     default:
145         break;
146 }
147
148
149 #ifdef HYPER_DEBUG
150 fprintf(stderr, "About to get into the read callback stuff\n");
151 #endif
152 if (ctx->amountRead < 5)
153 {
154     readData.length = 5 - ctx->amountRead;
155     readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
156     len = readData.length;
157     if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
158     {
159         if (err == SSLWouldBlockErr)
160             ctx->amountRead += len;
161         else if (err == SSLIOClosedOverrideGoodbyeKiss && ctx->amountRead ==
162 0)
163         {
164             SSLClose(ctx);
165             return SSLConnectionClosedGraceful;
166         }
167         else
168             SSLFatalSessionAlert(alert_close_notify, ctx);
169         return err;
170     }
171     ctx->amountRead += len;
172 }
173
174 ASSERT(ctx->amountRead >= 5);
175
176 progress = ctx->partialReadBuffer.data;
177 rec->contentType = *progress++;
178 if (rec->contentType < SSL_smallest_3_0_type ||
179     rec->contentType > SSL_largest_3_0_type)
180     return ERR(SSLProtocolErr);
181
182 rec->protocolVersion = (SSLProtocolVersion)SSLDecodeInt(progress, 2);
183 progress += 2;
184 contentLen = SSLDecodeInt(progress, 2);
185 progress += 2;
186 if (contentLen > (16384 + 2048)) /* Maximum legal length of an SSLCipherText payload */
187 {
188     SSLFatalSessionAlert(alert_unexpected_message, ctx);
189     return ERR(SSLProtocolErr);
190 }
191
192 if (ctx->partialReadBuffer.length < 5 + contentLen)
193 {
194     if ((err = SSLReallocBuffer(&ctx->partialReadBuffer, 5 + contentLen, &ctx->sysCtx)) !=
195 0)
196     {
197         SSLFatalSessionAlert(alert_close_notify, ctx);
198         return ERR(err);
199     }
200 }
201
202 if (ctx->amountRead < 5 + contentLen)
203 {
204     readData.length = 5 + contentLen - ctx->amountRead;
205     readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
206     len = readData.length;
207     if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
208     {
209         if (err == SSLWouldBlockErr)
210             ctx->amountRead += len;
211         else
212             SSLFatalSessionAlert(alert_close_notify, ctx);
213         return err;
214     }
215     ctx->amountRead += len;
216 }
217 }
```

```
208     ASSERT(ctx->amountRead >= 5 + contentLen);
209
210     cipherFragment.data = ctx->partialReadBuffer.data + 5;
211     cipherFragment.length = contentLen;
212
213     /* Decrypt the payload & check the MAC, modifying the length of the buffer to indicate the
214      * amount of plaintext data after adjusting for the block size and removing the MAC
215      * (this function generates its own alerts)
216      */
217     if ((err = DecryptSSLRecord(rec->contentType, &cipherFragment, ctx)) != 0)
218         return err;
219
220     /* We appear to have successfully received a record; increment the sequence number */
221     if (rec->contentType != SSL_application_data_sslappy)
222         IncrementUInt64(&ctx->readCipher.sequenceNum);
223
224
225 #ifdef SSL_COMPRESSION
226     if ((ctx->compressNow) && (ctx->selectedCompression != NULL) &&
227         (ctx->selectedCompression->identifier != 0)) {
228
229         /* Allocate a buffer to return the plaintext in and return it */
230         if ((err = SSLAllocBuffer(&rec->contents, DEFAULT_BUFFER_SIZE,
231
232             &ctx->sysCtx)) != SSLNoErr) {
233             SSLFatalSessionAlert(alert_close_notify, ctx);
234             return ERR(err);
235         }
236         if ((err = ctx->selectedCompression->process(cipherFragment,
237
238             &(rec->contents),
239
240             ctx->readCompressRef,
241
242             ctx)) != SSLNoErr) {
243             SSLFreeBuffer(&rec->contents, &ctx->sysCtx);
244             SSLFatalSessionAlert(alert_decompression_failure, ctx);
245             return ERR(err);
246         }
247     }
248 #endif
249
250     fprintf(stderr, "Decompression created output of %d from size %d\n",
251             cipherFragment.length,
252             rec->contents.length);
253
254 #endif
255     } else {
256         if ((err = SSLAllocBuffer(&rec->contents, cipherFragment.length,
257
258             &ctx->sysCtx)) != 0)
259         {
260             SSLFatalSessionAlert(alert_close_notify, ctx);
261             return ERR(err);
262         }
263         memcpy(rec->contents.data, cipherFragment.data, (size_t)
264             cipherFragment.length);
265     }
266 #else
267     memcpy(rec->contents.data, cipherFragment.data, (size_t) cipherFragment.length);
268 #endif
269
270     ctx->amountRead = 0;          /* We've used all the data in the cache */
271
272     return SSLNoErr;
273 }
274
275 /* SSLWriteRecord does not send alerts on failure, out of the assumption/fear
276  * that this might result in a loop (since sending an alert causes SSLWriteRecord
277  * to be called).
278  */
```

```

269 SSLerr
270 SSLWriteRecord(SSLRecord rec, SSLContext *ctx)
271 {
272     SSLerr      err;
273     int          padding = 0, i, freerec = 0;
274     WaitingRecord *out, *queue;
275     SSLBuffer     buf, payload, secret, mac, nonce;
276     uint8         *progress;
277     uint16        payloadSize, blockSize, nonceSize = 0;
278
279     if (rec.protocolVersion == SSL_Version_2_0)
280         return SSL2WriteRecord(rec, ctx);
281
282     ASSERT(rec.protocolVersion == SSL_Version_3_0);
283     ASSERT(rec.contents.length <= 16384);
284
285 #ifdef SSL_COMPRESSION
286     if((ctx->compressNow) && (ctx->selectedCompression != NULL) &&
287         (ctx->selectedCompression->identifier != 0)) {
288         SSLBuffer compdata;
289
290         /* make a guess about how long the buffer will need to be */
291         if((err = SSLAllocBuffer(&compdata, rec.contents.length + 4,
292             &ctx->sysCtx)) != SSLNoErr)
293             return ERR(err);
294         if((err = ctx->selectedCompression->process(rec.contents, &compdata,
295
296             ctx->writeCompressRef,
297
298             ctx)) != SSLNoErr) {
299             SSLFreeBuffer(&compdata, &ctx->sysCtx);
300             return ERR(err);
301         }
302         rec.contents = compdata;
303         freerec = 1;
304     }
305 #endif
306
307     out = 0;
308     /* Allocate a WaitingRecord to store our ready-to-send record in */
309     if ((err = SSLAllocBuffer(&buf, sizeof(WaitingRecord), &ctx->sysCtx)) != 0)
310         return ERR(err);
311     out = (WaitingRecord*)buf.data;
312     out->next = 0;
313     out->sent = 0;
314
315     /* Allocate enough room for the transmitted record, which will be:
316     * 5 bytes of header +
317     * encrypted contents +
318     * macLength +
319     * padding [block ciphers only] +
320     * padding length field (1 byte) [block ciphers only]
321     */
322     payloadSize = (uint16) (rec.contents.length + ctx->writeCipher.hash->digestSize);
323     blockSize = ctx->writeCipher.symCipher->blockSize;
324     if (blockSize > 0)
325     {
326         padding = blockSize - (payloadSize % blockSize) - 1;
327         payloadSize = (uint16) (payloadSize + padding + 1);
328     }
329
330     if(ctx->ssloppy)
331     {
332         /* in this case we need more room, for the nonce */
333         nonceSize = (uint16) MAX(sizeof(uint64), ctx->writeCipher.symCipher->ivSize);
334         payloadSize += nonceSize; decided this was wrong logic */
335     }
336
337     out->data.data = 0;

```

```
334     if ((err = SSLAllocBuffer(&out->data, 5 + payloadSize + nonceSize,
335                               &ctx->sysCtx)) != 0)
336         goto fail;
337
338     progress = out->data.data;
339     *(progress++) = rec.contentType;
340     progress = SSLEncodeInt(progress, rec.protocolVersion, 2);
341     progress = SSLEncodeInt(progress, payloadSize, 2);
342
343     /* Copy the contents into the output buffer */
344     memcpy(progress, rec.contents.data, (size_t) rec.contents.length);
345     payload.data = progress;
346     payload.length = rec.contents.length;
347
348     progress += rec.contents.length;
349     /* MAC immediately follows data */
350     mac.data = progress;
351     mac.length = ctx->writeCipher.hash->digestSize;
352     progress += mac.length;
353
354     if(ctx->ssloppy)
355     {
356         uint64 noncevalue;
357
358         if((err = SSLAllocBuffer(&nonce, nonceSize, &ctx->sysCtx)) != SSLNoErr)
359             goto fail;
360         if((err = ctx->sysCtx.random(nonce, ctx->sysCtx.randomRef)) != SSLNoErr)
361             goto fail;
362
363         memcpy(&noncevalue, nonce.data, sizeof(noncevalue));
364
365         /* MAC the data, sloppy-style */
366         if (mac.length > 0) /* Optimize away null case */
367         {
368             secret.data = ctx->writeCipher.macSecret;
369             secret.length = ctx->writeCipher.hash->digestSize;
370             if ((err = ComputeMAC(rec.contentType, payload, mac, noncevalue,
371                                 secret, &ctx->writeCipher, ctx)) != 0)
372                 goto fail;
373         }
374
375         memcpy(progress, nonce.data, nonce.length);
376         progress += nonce.length;
377     }
378     else
379     {
380         /* MAC the data, normal mode */
381         if (mac.length > 0) /* Optimize away null case */
382         {
383             secret.data = ctx->writeCipher.macSecret;
384             secret.length = ctx->writeCipher.hash->digestSize;
385             if ((err = ComputeMAC(rec.contentType, payload, mac,
386                                 ctx->writeCipher.sequenceNum, secret,
387                                 &ctx->writeCipher, ctx)) != 0)
388                 goto fail;
389         }
390     }
391 }
392
393 /* Update payload to reflect encrypted data: contents, mac & padding */
394 payload.length = payloadSize;
395
396 /* Fill in the padding bytes & padding length field with the padding value; the
397  * protocol only requires the last byte,
398  * but filling them all in avoids leaking data
399  */
400 if (ctx->writeCipher.symCipher->blockSize > 0)
401     for (i = 1; i <= padding + 1; ++i)
402         payload.data[payload.length - i] = (uint8)padding;
403
404 /* Encrypt the data */
```

Revision 1.6.10.1, by *marcvh*

```

405     DUMP_BUFFER_NAME("cleartext data", payload);
406     if ((err = ctx->writeCipher.symCipher->encrypt(payload, payload,
407
                                                    ctx-
>ssloppy ? &nonce:NULL,
408
                                                    ctx-
>writeCipher.symCipherState,
409
                                                    ctx))
        != 0)
410         goto fail;
411     DUMP_BUFFER_NAME("encrypted data", payload);
412
413     /* Enqueue the record to be written from the idle loop */
414     if (ctx->recordWriteQueue == 0)
415         ctx->recordWriteQueue = out;
416     else
417     {
418         queue = ctx->recordWriteQueue;
419         while (queue->next != 0)
420             queue = queue->next;
421         queue->next = out;
422     }
423
424     if(ctx->ssloppy)
425         SSLFreeBuffer(&nonce, &ctx->sysCtx);
426     else
427         /* Increment the sequence number */
428         IncrementUInt64(&ctx->writeCipher.sequenceNum);
429
430     if(freerec)
431         SSLFreeBuffer(&(rec.contents), &ctx->sysCtx);
432
433     return SSLNoErr;
434
435 fail: /* Only for if we fail between when the WaitingRecord is allocated and when it is
        queued */
436     SSLFreeBuffer(&out->data, &ctx->sysCtx);
437     buf.data = (uint8*)out;
438     buf.length = sizeof(WaitingRecord);
439     SSLFreeBuffer(&buf, &ctx->sysCtx);
440     if(freerec)
441         SSLFreeBuffer(&(rec.contents), &ctx->sysCtx);
442     return ERR(err);
443 }
444
445 static SSLErr
446 DecryptSSLRecord(uint8 type, SSLBuffer *payload, SSLContext *ctx)
447 {
448     SSLErr err;
449     SSLBuffer content, nonce;
450
451     if(type == SSL_application_data_ssloppy)
452     {
453         nonce.length = MAX(sizeof(uint64), ctx->readCipher.symCipher->ivSize);
454         nonce.data = payload->data + (payload->length - nonce.length);
455         payload->length -= nonce.length;
456     }
457
458     if ((ctx->readCipher.symCipher->blockSize > 0) &&
459         ((payload->length % ctx->readCipher.symCipher->blockSize) != 0))
460     {
461         SSLFatalSessionAlert(alert_unexpected_message, ctx);
462         return ERR(SSLProtocolErr);
463     }
464
465     /* Decrypt in place */
466     DUMP_BUFFER_NAME("encrypted data", (*payload));
467
468     if(type == SSL_application_data_ssloppy)
469     {
470         if ((err = ctx->readCipher.symCipher->decrypt(*payload, *payload, &nonce, ctx-
471 >readCipher.symCipherState, ctx)) != 0)
472         {

```

```
470             SSLFatalSessionAlert(alert_close_notify, ctx);
471             return ERR(err);
472         }
473     }
474     else
475     {
476         if ((err = ctx->readCipher.symCipher->decrypt(*payload, *payload, NULL, ctx-
477 >readCipher.symCipherState, ctx)) != 0)
478         {
479             SSLFatalSessionAlert(alert_close_notify, ctx);
480             return ERR(err);
481         }
482     }
483     DUMP_BUFFER_NAME("decrypted data", (*payload));
484     /* Locate content within decrypted payload */
485     content.data = payload->data;
486     content.length = payload->length - ctx->readCipher.hash->digestSize;
487     if (ctx->readCipher.symCipher->blockSize > 0)
488     {
489         /* padding can't be equal to or more than a block */
490         if (payload->data[payload->length - 1] >= ctx->readCipher.symCipher->blockSize)
491         {
492             SSLFatalSessionAlert(alert_unexpected_message, ctx);
493             return ERR(SSLProtocolErr);
494         }
495         content.length -= 1 + payload->data[payload->length - 1]; /* Remove block size
496         padding */
497     }
498     /* Verify MAC on payload */
499     if (ctx->readCipher.hash->digestSize > 0) /* Optimize away MAC for null case */
500     {
501         if (type == SSL_application_data_ssloppy)
502         {
503             uint64 nonceNumber;
504             memcpy(&nonceNumber, nonce.data, sizeof(nonceNumber));
505             if ((err = VerifyMAC(type, content, payload->data + content.length,
506             nonceNumber, ctx)) != 0)
507             {
508                 SSLFatalSessionAlert(alert_bad_record_mac, ctx);
509                 return ERR(err);
510             }
511         }
512         else
513         {
514             if ((err = VerifyMAC(type, content, payload->data + content.length,
515             ctx->readCipher.sequenceNum, ctx)) !=
516             0)
517             {
518                 SSLFatalSessionAlert(alert_bad_record_mac, ctx);
519                 return ERR(err);
520             }
521         }
522     }
523     *payload = content; /* Modify payload buffer to indicate content length */
524     return SSLNoErr;
525 }
526 static uint8*
527 SSLEncodeUInt64(uint8 *p, uint64 value)
528 {
529     p = SSLEncodeInt(p, value.high, 4);
530     return SSLEncodeInt(p, value.low, 4);
531 }
532 static SSLerr
533 VerifyMAC(uint8 type, SSLBuffer data, uint8 *compareMAC, uint64 seqNo, SSLContext *ctx)
534 {
535     SSLerr err;
536     uint8 macData[MAX_DIGEST_SIZE];
537     SSLBuffer secret, mac;
```

Revision 1.6.10.1, by *marcvh*

```

538     secret.length = ctx->readCipher.hash->digestSize;
539     mac.data = macData;
540     mac.length = ctx->readCipher.hash->digestSize;
541
542     if ((err = ComputeMAC(type, data, mac, seqNo, secret,
543         &ctx->readCipher, ctx)) != 0)
544         return ERR(err);
545
546     if ((memcmp(mac.data, compareMAC, (size_t) mac.length)) != 0)
547         return ERR(SSLProtocolErr);
548
549     return SSLNoErr;
550 }
551
552 static SSLerr
553 ComputeMAC(uint8 type, SSLBuffer data, SSLBuffer mac, uint64 seqNo, SSLBuffer secret,
554     CipherContext *cipherCtx, SSLContext *ctx)
555 {
556     SSLerr      err;
557     uint8       innerDigestData[MAX_DIGEST_SIZE];
558     uint8       scratchData[11], *progress;
559     SSLBuffer    digest, scratch;
560
561 #ifdef HYPER_DEBUG
562     int i;
563     fprintf(stderr, "Buffer: ");
564     for(i = 0; i < data.length; i++)
565         fprintf(stderr, "%02x ", data.data[i]);
566     fprintf(stderr, "\n");
567     fprintf(stderr, "sequenceno: ");
568     for(i = 0; i < sizeof(uint64); i++)
569         fprintf(stderr, "%02x ", (unsigned char) *((unsigned char *) &seqNo) + i);
570     fprintf(stderr, "\n");
571     fprintf(stderr, "Secret: ");
572     for(i = 0; i < secret.length; i++)
573         fprintf(stderr, "%02x ", secret.data[i]);
574     fprintf(stderr, "\n");
575 #endif
576
577     ASSERT(cipherCtx->hash->macPadSize <= MAX_MAC_PADDING);
578     ASSERT(cipherCtx->hash->digestSize <= MAX_DIGEST_SIZE);
579     ASSERT(SSLMACPad1[0] == 0x36 && SSLMACPad2[0] == 0x5C);
580
581     if(cipherCtx->digestCtx.data == NULL) {
582         if ((err = SSLAllocBuffer(&cipherCtx->digestCtx,
583             cipherCtx->hash->contextSize, &ctx->sysCtx))
584             != 0)
585             goto exit;
586         cipherCtx->hash->create(cipherCtx->digestCtx);
587     }
588
589     /* MAC = hash( MAC_write_secret + pad_2 + hash( MAC_write_secret + pad_1 + seq_num + type +
590     length + content ) ) */
591     if ((err = cipherCtx->hash->init(cipherCtx->digestCtx)) != 0)
592         goto exit;
593     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, secret)) != 0) /* MAC secret */
594         goto exit;
595     scratch.data = SSLMACPad1;
596     scratch.length = cipherCtx->hash->macPadSize;
597     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0) /* pad1 */
598         goto exit;
599     progress = scratchData;
600     progress = SSLEncodeUInt64(progress, seqNo);
601     *progress++ = type;
602     progress = SSLEncodeInt(progress, data.length, 2);
603     scratch.data = scratchData;
604     scratch.length = 11;
605     ASSERT(progress == scratchData+11);
606     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0) /* sequenceNo,
type & length */

```

```
607         goto exit;
608     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, data)) != 0) /* content */
609         goto exit;
610     digest.data = innerDigestData;
611     digest.length = cipherCtx->hash->digestSize;
612     if ((err = cipherCtx->hash->final(cipherCtx->digestCtx, digest)) != 0) /* figure inner
digest */
613         goto exit;
614
615     if ((err = cipherCtx->hash->init(cipherCtx->digestCtx)) != 0)
616         goto exit;
617     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, secret)) != 0) /* MAC secret */
618         goto exit;
619     scratch.data = SSLMACPad2;
620     scratch.length = cipherCtx->hash->macPadSize;
621     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0) /* pad2 */
622         goto exit;
623     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, digest)) != 0) /* inner digest
*/
624         goto exit;
625     if ((err = cipherCtx->hash->final(cipherCtx->digestCtx, mac)) != 0) /* figure the mac */
626         goto exit;
627
628     err = SSLNoErr; /* redundant, I know */
629
630 exit:
631     return ERR(err);
632 }
```


EXHIBIT D

Revision 1.11.2.1, by *marcvh*

```

0  /* *****
1  File: ciphers.c
2
3  SSL Plus: Security Integration Suite(tm)
4  Version 1.1.1 -- August 11, 1997
5
6  Copyright (c)1996, 1997 by Consensus Development Corporation
7      Copyright (c)1997, 1998 by Aventail Corporation
8
9  Portions of this software are based on SSLRef(tm) 3.0, which is
10 Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
11 was developed by Netscape Communications Corporation and Consensus
12 Development Corporation.
13
14 In order to obtain this software, your company must have signed
15 either a PRODUCT EVALUATION LICENSE (a copy of which is included in
16 the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
17 licenses have different limitations regarding how you are allowed to
18 use the software. Before retrieving (or using) this software, you
19 *must* ascertain which of these licenses your company currently
20 holds. Then, by retrieving (or using) this software you agree to
21 abide by the particular terms of that license. If you do not agree
22 to abide by the particular terms of that license, than you must
23 immediately delete this software. If your company does not have a
24 signed license of either kind, then you must either contact
25 Consensus Development and execute a valid license before retrieving
26 (or using) this software, or immediately delete this software.
27
28 *****
29
30 File: ciphers.c      Data structures for handling supported ciphers
31
32 Contains a table mapping cipherSuite values to the ciphers, MAC
33 algorithms, key exchange procedures and so on that are used for that
34 algorithm, in order of preference.
35
36 ***** */
37
38 #ifndef _CRYPTYPE_H
39 #include <cryptype.h>
40 #endif
41
42 #ifndef _SSLCTX_H
43 #include <sslctx.h>
44 #endif
45
46 #include <string.h>
47
48 extern SSLSymmetricCipher SSLCipherNull;
49 extern SSLSymmetricCipher SSLCipherDES_CBC;
50 extern SSLSymmetricCipher SSLCipherDES40_CBC;
51 extern SSLSymmetricCipher SSLCipherRC4_40;
52 extern SSLSymmetricCipher SSLCipherRC4_56;
53 extern SSLSymmetricCipher SSLCipherRC4_128;
54 extern SSLSymmetricCipher SSLCipher3DES_CBC;
55
56 /* Even if we don't support NULL_WITH_NULL_NULL for transport, we need a reference for startup
57 */
58 SSLCipherSpec SSL_NULL_WITH_NULL_NULL_CipherSpec =
59 {
60     SSL_NULL_WITH_NULL_NULL,
61     Exportable,
62     SSL_NULL_auth,
63     &SSLHashNullOpt,
64     &SSLCipherNull
65 };
66
67 /* Disable non-exportable cipher suites to build an export only library */
68 #ifndef ENABLE_NONEXPORT_CIPHERS
69 #define ENABLE_NONEXPORT_CIPHERS 1
70 #endif
71
72

```

```
70 /* Disable exportable cipher suites to build a strong crypto only library */
71 #ifndef ENABLE_EXPORT_CIPHERS
72 #define ENABLE_EXPORT_CIPHERS 1
73 #endif
74
75 /* Reenable DH-anon only if you know you want to use Diffie-Hellman cipher suites:
76    Enabling DH-anon leaves you open to a man-in-the-middle attack which can degrade your
77    security to this level. */
78 #ifndef ENABLE_DH_ANON
79 #define ENABLE_DH_ANON 0
80 #endif
81
82 /* Reenable NULL encryption cipher suites only if you know for a fact you want to support
83    unencrypted sessions. Unencrypted sessions do not provide data privacy and may be more
84    vulnerable to attack than encrypted sessions. */
85 #ifndef ENABLE_NULL_CIPHERS
86 #define ENABLE_NULL_CIPHERS 0
87 #endif
88
89 #ifdef VIRGIN_SSLPLUS
90 /* Order by preference */
91 SSLCipherSpec KnownCipherSpecs[] =
92 {
93 #if ENABLE_NONEXPORT_CIPHERS
94     { SSL_RSA_WITH_3DES_EDE_CBC_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipher3DES_CBC
95     },
96     { SSL_RSA_WITH_RC4_128_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipherRC4_128 },
97     { SSL_RSA_WITH_RC4_128_MD5, NotExportable, SSL_RSA, &SSLHashMD5, &SSLCipherRC4_128 },
98     { SSL_RSA_WITH_DES_CBC_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipherDES_CBC },
99 #endif
100 #if ENABLE_EXPORT_CIPHERS
101     { SSL_RSA_EXPORT_WITH_RC4_40_MD5, Exportable, SSL_RSA_EXPORT, &SSLHashMD5,
102     &SSLCipherRC4_40 },
103     { SSL_RSA_EXPORT_WITH_DES40_CBC_SHA, Exportable, SSL_RSA_EXPORT, &SSLHashSHA1,
104     &SSLCipherDES40_CBC },
105 #endif
106 #if ENABLE_DH_ANON && ENABLE_NONEXPORT_CIPHERS
107     { SSL_DH_anon_WITH_3DES_EDE_CBC_SHA, NotExportable, SSL_DH_anon, &SSLHashSHA1,
108     &SSLCipher3DES_CBC },
109     { SSL_DH_anon_WITH_RC4_128_MD5, NotExportable, SSL_DH_anon, &SSLHashMD5,
110     &SSLCipherRC4_128 },
111     { SSL_DH_anon_WITH_DES_CBC_SHA, NotExportable, SSL_DH_anon, &SSLHashSHA1,
112     &SSLCipherDES_CBC },
113 #endif
114 #if ENABLE_NULL_CIPHERS && ENABLE_EXPORT_CIPHERS
115     { SSL_RSA_WITH_NULL_SHA, Exportable, SSL_RSA, &SSLHashSHA1, &SSLCipherNull },
116     { SSL_RSA_WITH_NULL_MD5, Exportable, SSL_RSA, &SSLHashMD5, &SSLCipherNull }
117 #endif
118 };
119
120 int CipherSpecCount = sizeof(KnownCipherSpecs) / sizeof(SSLCipherSpec);
121 #endif /* VIRGIN_SSLPLUS */
122
123 SSLerr
124 FindCipherSpec(SSLContext *ctx, uint16 specID, SSLCipherSpec* *spec)
125 {
126     int i;
127     uint32 mask;
128
129     *spec = 0;
130     for (i = 0; i < CipherSpecCount; i++)
131     {
132         if (KnownCipherSpecs[i].cipherSpec == specID)
133         {
134             mask = (uint32) 1;
135             mask <= i;
136             if (ctx->cipherspecs & mask)
137             {
138                 *spec = &KnownCipherSpecs[i];
139                 break;
140             }
141         }
142     }
143 }
```

```

135     }
136 }
137
138     if (*spec == 0)          /* Not found */
139         return SSLNegotiationErr;
140     return SSLNoErr;
141 }
142
143 SSLerr SSLDESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx);
144 SSLerr SSLDESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
    *ctx);
145 SSLerr SSLDESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
    *ctx);
146 SSLerr SSLDESFinish(void *cipherRef, SSLContext *ctx);
147 SSLerr SSLDESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob);
148 SSLerr SSLDESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob);
149
150 SSLSymmetricCipher SSLCipherDES_CBC = {
151     8,          /* Key size in bytes */
152     8,          /* Secret key size = 64 bits */
153     8,          /* IV size */
154     8,          /* Block size */
155     SSLDESInit,
156     SSLDESEncrypt,
157     SSLDESDecrypt,
158     SSLDESFinish,
159     SSLDESExport,
160     SSLDESImport
161 };
162
163 SSLSymmetricCipher SSLCipherDES40_CBC = {
164     8,          /* Key size in bytes */
165     5,          /* Secret key size = 40 bits */
166     8,          /* IV size */
167     8,          /* Block size */
168     SSLDESInit,
169     SSLDESEncrypt,
170     SSLDESDecrypt,
171     SSLDESFinish
172 };
173
174 typedef struct _DESState
175 {
176     unsigned char key[24]; /* work for 3DES and DES both */
177     unsigned char iv[8];
178     int reading; /* do we really need this? */
179     B_ALGORITHM_OBJ des;
180 } DESState;
181
182 SSLerr
183 SSLDESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx)
184 {
185     SSLBuffer          desState;
186     B_ALGORITHM_OBJ    *des;
187     static B_ALGORITHM_METHOD *chooser[] = { &AM_DES_CBC_ENCRYPT, &AM_DES_CBC_DECRYPT, 0 };
188     B_KEY_OBJ          desKey;
189     ITEM               keyData;
190     SSLerr             err;
191     int                rsaErr;
192     DESState *s;
193
194     if ((err = SSLAllocBuffer(&desState, sizeof(DESState), &ctx->sysCtx)) != 0)
195         return err;
196     s = (DESState *)desState.data;
197
198     memcpy(s->key, key, 8);
199     memcpy(s->iv, iv, 8);
200
201     if ((rsaErr = B_CreateAlgorithmObject(&(s->des))) != 0)
202         return SSLUnknownErr;

```

Revision 1.11.2.1, by *marcvh*

```

203     if ((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, iv)) != 0)
204         return SSLUnknownErr;
205     if ((rsaErr = B_CreateKeyObject(&desKey)) != 0)
206         return SSLUnknownErr;
207     keyData.data = key;
208     keyData.len = 8;
209     if ((rsaErr = B_SetKeyInfo(desKey, KI_DES8, key)) != 0)
210     {
211         B_DestroyKeyObject(&desKey);
212         return SSLUnknownErr;
213     }
214     if (cipherRef == (void*)&(ctx->writePending.symCipherState))
215     {
216         s->reading = 0;
217         if ((rsaErr = B_EncryptInit(*des, desKey, chooser, &ctx->sysCtx.yield)) != 0)
218         {
219             B_DestroyKeyObject(&desKey);
220             return SSLUnknownErr;
221         }
222     }
223     else if (cipherRef == (void*)&(ctx->readPending.symCipherState))
224     {
225         s->reading = 1;
226         if ((rsaErr = B_DecryptInit(*des, desKey, chooser, &ctx->sysCtx.yield)) != 0)
227         {
228             B_DestroyKeyObject(&desKey);
229             return SSLUnknownErr;
230         }
231     }
232     else
233         ASSERTMSG("Couldn't determine read/writeness");
234     B_DestroyKeyObject(&desKey);
235     *cipherRef = (void*)s;
236     return SSLNoErr;
237 }
238
239 SSLErr
240 SSLDESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
241 {
242     DESState *s = (DESState *) cipherRef;
243     void *subCipherRef = NULL;
244     int      rsaErr;
245     unsigned int  outputLen;
246     SSLBuffer    temp;
247     SSLErr      err;
248
249     if(cipherRef == NULL)
250         return SSLUnknownErr;
251
252     if(iv != NULL)
253     {
254         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8,
255                                         (POINTER) iv->data)) !=
256         SSLNoErr)
257             return err;
258     }
259     else
260     {
261         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, s->iv)) != SSLNoErr)
262             return err;
263     }
264
265     ASSERT(src.length == dest.length);
266     ASSERT(src.length % 8 == 0);
267
268     if (src.data == dest.data)
269     /* BSAFE won't let you encrypt in place */
270     {
271         if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
272             return err;
273         memcpy(temp.data, src.data, (size_t) src.length);
274     }

```

```
273     else
274         temp = src;
275
276     if ((rsaErr = B_EncryptUpdate(s->des, dest.data, &outputLen,
277         (unsigned int) dest.length, temp.data,
278         (unsigned int) temp.length,
279         (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
280     {
281         if (src.data == dest.data)
282             SSLFreeBuffer(&temp, &ctx->sysCtx);
283         return SSLUnknownErr;
284     }
285     ASSERT(outputLen == src.length);
286
287     if (src.data == dest.data)
288         SSLFreeBuffer(&temp, &ctx->sysCtx);
289
290     if (outputLen != src.length)
291         return SSLUnknownErr;
292
293     /* if not doing SSLoppy, save the IV for next time... */
294     if(iv == NULL)
295     {
296         unsigned char *buf;
297
298         if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
299             AI_DES_CBC_IV8))
300             != SSLNoErr)
301             return err;
302
303         memcpy(s->iv, buf, sizeof(s->iv));
304     }
305
306     /* memcpy(s->iv, dest.data + dest.length - 8, 8); */
307
308     return SSLNoErr;
309 }
310
311 SSLErr
312 SSLDESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
313 {
314     DESState *s = (DESState *) cipherRef;
315     int          rsaErr;
316     unsigned int  outputLen;
317     SSLBuffer     temp;
318     SSLErr        err;
319
320     if(cipherRef == NULL)
321         return SSLUnknownErr;
322
323     if(iv != NULL)
324     {
325         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, (POINTER) iv->data))
326             != SSLNoErr)
327             return err;
328     }
329     else
330     {
331         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, s->iv)) != SSLNoErr)
332             return err;
333     }
334
335     ASSERT(src.length == dest.length);
336     ASSERT(src.length % 8 == 0);
337
338     /* memcpy(s->iv, src.data + src.length - 8, 8); */
339
340     if (src.data == dest.data)
341     /* BSAFE won't let you encrypt in place */
342     {
343         if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
344             return err;
```

```

344     memcpy(temp.data, src.data, (size_t) src.length);
345 }
346 else
347     temp = src;
348
349 if ((rsaErr = B_DecryptUpdate(s->des, dest.data, &outputLen,
350                             (unsigned int) dest.length, temp.data,
351                             (unsigned int) temp.length,
352                             (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
353 {
354     if (src.data == dest.data)
355         SSLFreeBuffer(&temp, &ctx->sysCtx);
356     return SSLUnknownErr;
357 }
358
359 ASSERT(outputLen == src.length);
360
361 if (src.data == dest.data)
362     SSLFreeBuffer(&temp, &ctx->sysCtx);
363
364 if (outputLen != src.length)
365     return SSLUnknownErr;
366
367 /* if not doing SSLOppy, save the IV for next time... */
368 if(iv == NULL)
369 {
370     unsigned char *buf;
371
372     if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
373                                     AI_DES_CBC_IV8))
374        != SSLNoErr)
375         return err;
376     memcpy(s->iv, buf, sizeof(s->iv));
377 }
378
379 return SSLNoErr;
380 }
381
382 SSLState
383 SSLDESFinish(void *cipherRef, SSLContext *ctx)
384 {
385     DESState *s = (DESState *) cipherRef;
386     SSLBuffer desState;
387     SSLState err;
388
389     if(cipherRef == NULL)
390         return SSLUnknownErr;
391
392     B_DestroyAlgorithmObject(&(s->des));
393
394     memset(cipherRef, 0, sizeof(DESState));
395     desState.data = (unsigned char *)cipherRef;
396     desState.length = sizeof(DESState);
397
398     err = SSLFreeBuffer(&desState, &ctx->sysCtx);
399     return err;
400 }
401
402 SSLState SSLDESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob)
403 {
404     DESState *s = (DESState *) cipherRef;
405
406     if(cipherRef == NULL)
407         return SSLUnknownErr;
408
409     if(blob->length < (8 + 8))
410         return SSLMemoryErr;
411
412     memcpy(blob->data, s->key, 8);
413     memcpy(blob->data + 8, s->iv, 8);
414     /* memcpy(blob->data + 16, &(s->reading), sizeof(int)); */
415     blob->length = 16;

```

```
415
416     return SSLNoErr;
417 }
418
419 SSLERR SSLDESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob)
420 {
421     unsigned char *key, *iv;
422
423     if(blob == NULL)
424         return SSLUnknownErr;
425     if(blob->length < 16)
426         return SSLUnknownErr;
427
428     key = blob->data;
429     iv = blob->data + 8;
430
431     return SSLDESInit(key, iv, cipherRef, ctx);
432 }
433
434
435 SSLERR SSL3DESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx);
436 SSLERR SSL3DESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
    *ctx);
437 SSLERR SSL3DESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
    *ctx);
438 SSLERR SSL3DESFinish(void *cipherRef, SSLContext *ctx);
439 SSLERR SSL3DESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob);
440 SSLERR SSL3DESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob);
441
442 SSLSymmetricCipher SSLCipher3DES_CBC = {
443     24,      /* Key size in bytes */
444     24,      /* Secret key size = 192 bits */
445     8,       /* IV size */
446     8,       /* Block size */
447     SSL3DESInit,
448     SSL3DESEncrypt,
449     SSL3DESDecrypt,
450     SSL3DESFinish,
451     SSL3DESExport,
452     SSL3DESImport
453 };
454
455 SSLERR
456 SSL3DESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx)
457 {
458     SSLBuffer          desState;
459     DESState *s;
460     static B_ALGORITHM_METHOD *chooser[] = { &AM_DES_EDE3_CBC_ENCRYPT,
461
462     &AM_DES_EDE3_CBC_DECRYPT, 0 };
463     B_KEY_OBJ          desKey;
464     ITEM               keyData;
465     SSLERR             err;
466     int                rsaErr;
467
468     if ((err = SSLAllocBuffer(&desState, sizeof(DESState), &ctx->sysCtx)) != 0)
469         return err;
470     s = (DESState *)desState.data;
471     if ((rsaErr = B_CreateAlgorithmObject(&(s->des))) != 0)
472         return SSLUnknownErr;
473     if ((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, iv)) != 0)
474         return SSLUnknownErr;
475     memcpy(s->iv, iv, 8);
476
477     if ((rsaErr = B_CreateKeyObject(&desKey)) != 0)
478         return SSLUnknownErr;
479     keyData.data = key;
480     keyData.len = 24;
481     if ((rsaErr = B_SetKeyInfo(desKey, KI_24Byte, key)) != 0)
482     {
483         B_DestroyKeyObject(&desKey);
484     }
```



```
483     return SSLUnknownErr;
484 }
485 memcpy(s->key, key, 24);
486
487 if (cipherRef == (void*)&(ctx->writePending.symCipherState))
488 {
489     if ((rsaErr = B_EncryptInit(s->des, desKey, chooser,
490                                &ctx->sysCtx.yield)) != 0)
491     {
492         B_DestroyKeyObject(&desKey);
493         return SSLUnknownErr;
494     }
495 }
496 else if (cipherRef == (void*)&(ctx->readPending.symCipherState))
497 {
498     if ((rsaErr = B_DecryptInit(s->des, desKey, chooser,
499                                &ctx->sysCtx.yield)) != 0)
500     {
501         B_DestroyKeyObject(&desKey);
502         return SSLUnknownErr;
503     }
504 }
505 else
506     ASSERTMSG("Couldn't determine read/writeness");
507
508 B_DestroyKeyObject(&desKey);
509 *cipherRef = (void*)desState.data;
510 return SSLNoErr;
511 }
512
513 SSLerr
514 SSL3DESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
515 {
516     DESState *s = (DESState *) cipherRef;
517     int      rsaErr;
518     unsigned int outputLen;
519     SSLBuffer temp;
520     SSLerr    err;
521
522     ASSERT(src.length == dest.length);
523     ASSERT(src.length % 8 == 0);
524     if(cipherRef == NULL)
525         return SSLUnknownErr;
526
527     if(iv != NULL)
528     {
529         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8,
530                                         (POINTER) iv->data)) !=
531            SSLNoErr)
532             return err;
533     }
534     else
535     {
536         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, s->iv)) != SSLNoErr)
537             return err;
538     }
539
540     if (src.data == dest.data)
541     /* BSAFE won't let you encrypt in place */
542     {
543         if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
544             return err;
545         memcpy(temp.data, src.data, (size_t) src.length);
546     }
547     else
548         temp = src;
549
550     if ((rsaErr = B_EncryptUpdate(s->des, dest.data, &outputLen,
551                                   (unsigned int) dest.length, temp.data,
552                                   (unsigned int) temp.length,
553                                   (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
```

```
553     {   if (src.data == dest.data)
554         SSLFreeBuffer(&temp, &ctx->sysCtx);
555         return SSLUnknownErr;
556     }
557
558     ASSERT(outputLen == src.length);
559
560     if (src.data == dest.data)
561         SSLFreeBuffer(&temp, &ctx->sysCtx);
562
563     if (outputLen != src.length)
564         return SSLUnknownErr;
565
566     if(iv == NULL)
567     {
568         unsigned char *buf;
569
570         if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
571                                     AI_DES_EDE3_CBC_IV8))
572            != SSLNoErr)
573             return err;
574         memcpy(s->iv, buf, sizeof(s->iv));
575     }
576
577     /* memcpy(s->iv, dest.data + dest.length - 8, 8); */
578
579     return SSLNoErr;
580 }
581
582 SSLErr
583 SSL3DESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
584 {
585     DESState *s = (DESState *) cipherRef;
586     int      rsaErr;
587     unsigned int  outputLen;
588     SSLBuffer    temp;
589     SSLErr      err;
590
591     ASSERT(src.length == dest.length);
592     ASSERT(src.length % 8 == 0);
593     if(cipherRef == NULL)
594         return SSLNoErr;
595
596     if(iv != NULL)
597     {
598         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8,
599                                     (POINTER) iv->data)) !=
600            SSLNoErr)
601             return err;
602     }
603     else
604     {
605         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, s->iv)) != SSLNoErr)
606             return err;
607     }
608
609     /* memcpy(s->iv, src.data + src.length - 8, 8); */
610
611     if (src.data == dest.data)
612     /* BSAFE won't let you encrypt in place */
613     {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
614         return err;
615         memcpy(temp.data, src.data, (size_t) src.length);
616     }
617     else
618         temp = src;
619
620     if ((rsaErr = B_DecryptUpdate(s->des, dest.data, &outputLen,
621                                 (unsigned int) dest.length, temp.data,
622                                 (unsigned int) temp.length,
623                                 (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
```

```
623     {   if (src.data == dest.data)
624           SSLFreeBuffer(&temp, &ctx->sysCtx);
625           return SSLUnknownErr;
626     }
627
628     if(iv == NULL)
629     {
630         unsigned char *buf;
631
632         if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
633                                         AI_DES_EDE3_CBC_IV8)) !=
        SSLNoErr)
634             return err;
635         memcpy(s->iv, buf, sizeof(s->iv));
636     }
637
638     ASSERT(outputLen == src.length);
639
640     if (src.data == dest.data)
641         SSLFreeBuffer(&temp, &ctx->sysCtx);
642
643     if (outputLen != src.length)
644         return SSLUnknownErr;
645
646     return SSLNoErr;
647 }
648
649 SSLErr
650 SSL3DESFinish(void *cipherRef, SSLContext *ctx)
651 {
652     DESState *s = (DESState *) cipherRef;
653     SSLBuffer      desState;
654     SSLErr         err;
655
656     if(cipherRef == NULL)
657         return SSLUnknownErr;
658
659     B_DestroyAlgorithmObject(&(s->des));
660
661     memset(cipherRef, 0, sizeof(DESState));
662     desState.data = (unsigned char*)cipherRef;
663     desState.length = sizeof(DESState);
664     err = SSLFreeBuffer(&desState, &ctx->sysCtx);
665     return err;
666 }
667
668 SSLErr SSL3DESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob)
669 {
670     DESState *s = (DESState *) cipherRef;
671
672     if(cipherRef == NULL)
673         return SSLUnknownErr;
674
675     if(blob->length < (24 + 8))
676         return SSLMemoryErr;
677
678     memcpy(blob->data, s->key, 24);
679     memcpy(blob->data + 24, s->iv, 8);
680     blob->length = 32;
681
682     return SSLNoErr;
683 }
684
685 SSLErr SSL3DESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob)
686 {
687     unsigned char *key, *iv;
688
689     if(blob == NULL)
690         return SSLUnknownErr;
691     if(blob->length < 32)
692         return SSLUnknownErr;
```

```
693
694     key = blob->data;
695     iv = blob->data + 24;
696
697     return SSL3DESInit(key, iv, cipherRef, ctx);
698 }
```

EXHIBIT E

```
0 /* Copyright Aventail Corporation 1997-2000; All Rights Reserved */
1 /* sslenv.c is the SSL environment file; it defines functions used by
2    the SSL module as callbacks for managing mutexes, memory, I/O,
3    and user interaction. */
4
5 #include "sslmain.h"
6 #include "sslldap.h"
7 #include "ldapcert.h"
8 #include <aglobal.h>
9 #include <bSAFE.h>
10 #include <pkcs.h>
```

<due to the size of this file and the small portion of it which is relevant here, we include only the single function SSLEncode>

```
970 int FAR EXPORT SSLEncode(S5Packet *ibuf, S5Packet *obuf, int flag, void *handle)
971 {
972     S5SSLHandle *ref = handle;
973     S5SSLFlowConnection *conn = &(ref->conn);
974     SSLContext *ctx = ref->ctx;
975     uint32 ilen;
976     uint32 len = ibuf ? ibuf->len : 0;
977     int ssloppy = 0;
978
979 #ifdef HYPER_DEBUG
980     int i;
981 #endif
982     SSLerr err;
983     uint32 wrtp = 0;
984
985     if (flag & S5_STATEDUMP)
986     {
987         SSLBuffer block;
988         unsigned totalSize;
989         PSSLStateDump dump = (PSSLStateDump)obuf->data;
990
991         if (obuf->len < 4096)
992         {
993             obuf->len = 4096;
994             return ENCODE_BUFFER_TOO_SMALL;
995         }
996
997         // get the SSL state
998         //
999         if ((err = S5ExportContext(ctx, &block)) != SSLNoErr)
1000         {
1001             if (GlobalUpdate)
1002                 GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_ERROR,
1003                               IDS_SSL_EXPORTCONTEXTFAILED, err);
1004             return -1;
1005         }
1006
1007         // compute the total size of the data
1008         //
1009         totalSize = block.length + sizeof(SSLStateDump);
1010
1011         // validate the output buffer size
1012         //
1013         if (obuf->len < totalSize)
1014         {
1015             obuf->len = totalSize;
1016             SSLFreeBuffer(&block, &ctx->sysCtx);
1017             return ENCODE_BUFFER_TOO_SMALL;
1018         }
1019
1020         // put the SSLStateDump structure at the beginning of the output buffer
1021         //
1022         dump->SSLContext = ctx;
1023         dump->ContextSize = sizeof(SSLContext);
1024         dump->SSLState.data = (uint8 *) (dump+1);
```

```
1025         dump->SSLState.length = block.length;
1026
1027         // copy the SSL state to the output buffer
1028         //
1029         memcpy(dump->SSLState.data, block.data, block.length);
1030         obuf->len = totalSize;
1031         SSLFreeBuffer(&block, &ctx->sysCtx);
1032
1033         if (GlobalUpdate)
1034             GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1035                          IDS_SSL_EXPORTEDCONTEXT, obuf->len);
1036
1037         return 0;
1038     }
1039
1040     if(ref->endtime > 0)
1041         if(time((time_t *) NULL) >= ref->endtime) {
1042             if(GlobalUpdate)
1043                 GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1044                              IDS_SSL_LIFETIMEEXCEEDED);
1045
1046             return -1;
1047         }
1048
1049     SSLGetWritePendingSize(ctx, &wrtp);
1050 #ifdef HYPER_DEBUG
1051     if(wrtp)
1052         if(GlobalUpdate)
1053             GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1054                          IDS_SSL_BYTESPENDINGWRITE, wrtp);
1055 #endif
1056
1057     if(flag & S5_DATAGRAM)
1058         if(ref->ssloppy)
1059         {
1060             if((rt = SSLSetSloppyMode(ctx, 1)) != SSLNoErr)
1061             {
1062                 if(GlobalUpdate)
1063                     GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_WARNING,
1064                                  IDS_SSL_SSLOPPYMODEFAILED, rt);
1065
1066                 return -1;
1067             }
1068             ssloppy = 1;
1069         }
1070         else
1071         {
1072             /* UDP naked, baby! */
1073             #ifdef HYPER_DEBUG
1074                 if(GlobalUpdate)
1075                     GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1076                                  IDS_SSL_GOTDATAGRAM, flag, ibuf->len);
1077             #endif
1078
1079             #ifdef OPTIMIZE_UDP_NAKED
1080                 /* this is much cleaner and faster, but causes inconsistency in the
1081                  * API from the caller. Sigh. */
1082                 *obuf = *ibuf;
1083             #else
1084                 obuf->data = HeapAlloc(GetProcessHeap(), 0, ibuf->len);
1085                 obuf->data = malloc(ibuf->len);
1086             #endif
1087
1088             if(obuf->data == NULL) {
1089                 #ifdef HYPER_DEBUG
1090                     FPRINTF(stderr, _T("Returning error, buf data is null in
1091                                obufdata\n"));
1092                 #endif
1093
1094                 return SSLMemoryErr;
1095             }
1096
1097             memcpy(obuf->data, ibuf->data, ibuf->len);
1098         }
1099     }
```

```
1094             obuf->len = ibuf->len;
1095 #endif
1096             return ibuf->len;
1097     }
1098
1099
1100
1101     if(flag & S5_ENCODE) {
1102
1103 #ifdef HYPER_DEBUG
1104         if(GlobalUpdate)
1105             GlobalUpdate(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1106                         IDS_SSL_ENCODINGBYTES, ibuf->len);
1107 #ifndef AUTOSOCKS
1108         for(i = 0; i < ibuf->len; i++)
1109             FPRINTF(stderr, _T("%02x "), ibuf->data[i]);
1110             FPRINTF(stderr, _T("\n"));
1111 #endif
1112 #endif
1113
1114 #define SSL_MAX_ENCODE_SIZE 4096
1115
1116 #ifdef SSL_MAX_ENCODE_SIZE
1117         if(ibuf->len > SSL_MAX_ENCODE_SIZE) {
1118             conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_VERBOSE,
1119                                   IDS_SSL_MAXENCODESIZEEXCEEDED,
1120                                   ibuf->len, SSL_MAX_ENCODE_SIZE);
1121             ibuf->len = SSL_MAX_ENCODE_SIZE;
1122         }
1123 #endif
1124
1125         if(obuf->data != NULL) {
1126             if(obuf->len < (int) (ibuf->len + SSL_HEADLEN + 64 + wrtp)) {
1127                 conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_DEBUG,
1128                                         IDS_SSL_BUFFERTOOSHORT,
1129                                         obuf->len,
1130                                         (ibuf->len + SSL_HEADLEN
1131                                          + 64 + wrtp));
1132                 obuf->len = (int) ibuf->len + SSL_HEADLEN + 64 + wrtp;
1133                 if(ssloppy) SSLSetSloppyMode(ctx, 0);
1134                 return ENCODE_BUFFER_TOO_SMALL;
1135             }
1136             conn->writebuffer.data = obuf->data;
1137             conn->writebuffer.len = obuf->len;
1138             conn->writebuffer.off = SSL_HEADLEN;
1139             conn->writeflag = SSL_FLOW_WRITE_NOMAKEBUF;
1140         } else
1141             conn->writeflag = 0;
1142
1143         ilen = (uint32) ibuf->len;
1144         if((err = SSLWrite(ibuf->data, &ilen, ctx))) {
1145             conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1146                                   IDS_SSL_WRITEERROR,err);
1147             if(ssloppy) SSLSetSloppyMode(ctx, 0);
1148             return -1;
1149         }
1150
1151         if(conn->writebuffer.off > 0xFFFF) {
1152             conn->modctx.log.update(sslLogHandle,S5_LOG_MISC,S5_LOG_ERROR,
1153                                   IDS_SSL_PACKETTOOBIG,
1154                                   conn->writebuffer.off);
1155             if(ssloppy) SSLSetSloppyMode(ctx, 0);
1156             return -1;
1157         }
1158
1159         if(obuf->data != NULL) {
1160             /* Here we shift the semantics of writebuffer; off now points
1161              * to the beginning of the data, and len points to the end of
1162              * the data, not the length of the buffer, which we no longer
1163              * need to know since we don't be depositing anything new in it */
1164             obuf->len = conn->writebuffer.off;
```



```
1163         conn->writebuffer.len = conn->writebuffer.off;
1164         conn->writebuffer.off = SSL_HEADLEN;
1165     } else {
1166         if(conn->writebuffer.off == SSL_HEADLEN)
1167             /* Wow! Some thoughtful soul in SSLFlowWrite has left us a 4
1168              byte offset in the writebuffer so we can insert our header
1169              without needing to malloc a new buffer and memcpy into it! */
1170             obuf->data = conn->writebuffer.data;
1171         else {
1172             #ifndef _WINDOWS
1173                 obuf->data = malloc(conn->writebuffer.len + SSL_HEADLEN);
1174             #else
1175                 #ifdef WIN32
1176                     obuf->data = HeapAlloc(GetProcessHeap(), 0,
1177                                             conn->writebuffer.len +
1178                                             SSL_HEADLEN);
1179                 #endif
1180             }
1181             if(obuf->data == NULL) {
1182                 conn->modctx.log.update(sslLogHandle, S5_LOG_MISC,
1183                                         S5_LOG_ERROR, IDS_SSL_MALLOCFAILED);
1184                 if(ssloppy) SSLSetSloppyMode(ctx, 0);
1185                 return -1;
1186             }
1187             memcpy(obuf->data + SSL_HEADLEN,
1188                   conn->writebuffer.data + conn->writebuffer.off,
1189                   conn->writebuffer.len - conn->writebuffer.off);
1190             #ifdef WIN32
1191                 HeapFree(GetProcessHeap(), 0, conn->writebuffer.data);
1192             #else
1193                 free(conn->writebuffer.data);
1194             #endif
1195         }
1196         obuf->len = (int) (conn->writebuffer.len -
1197                           conn->writebuffer.off + SSL_HEADLEN);
1198     }
1199
1200     obuf->data[0] = SSL_HEADVERSION;
1201     obuf->data[1] = conn->state;
1202     obuf->data[2] = (uint8) ((conn->writebuffer.len - conn->writebuffer.off)
1203                             >> 8);
1204     obuf->data[3] = (uint8) ((conn->writebuffer.len - conn->writebuffer.off)
1205                             & 0xFF);
1206     conn->writebuffer.data = NULL;
1207     conn->writebuffer.len = 0;
1208     conn->writebuffer.off = 0;
1209     if(GlobalUpdate)
1210         GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1211                     IDS_SSL_ENCODERRETURNING, obuf->len, ilen);
1212 #ifdef HYPER_DEBUG
1213     for(i = 0; i < obuf->len; i++)
1214         FPRINTF(stderr, _T("%02x "), obuf->data[i]);
1215     FPRINTF(stderr, _T("\n"));
1216 #endif
1217 #endif
1218     if(ssloppy) SSLSetSloppyMode(ctx, 0);
1219     return (int) ilen;
1220 }
1221
1222 /* we must be decoding, instead of encoding.. */
1223 #ifdef HYPER_DEBUG
1224     if(GlobalUpdate)
1225         GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1226                     IDS_SSL_DECODINGBYTES, ibuf->len);
1227     if(GlobalUpdate)
1228         GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1229                     IDS_SSL_READBUFFERCOMINGIN,
```

```
1232                                     conn->readbuffer.len - conn->readbuffer.off);
1233 #ifndef AUTOSOCKS
1234     for(i = 0; i < ibuf->len; i++)
1235         FPRINTF(stderr, _T("%02x "), ibuf->data[i]);
1236     FPRINTF(stderr, _T("\n"));
1237 #endif
1238 #endif
1239
1240     if(ibuf->len < SSL_HEADLEN)
1241     {
1242         conn->modctx.log.update(sslLogHandle, S5_LOG_MISC, S5_LOG_ERROR,
1243                               IDS_SSL_DECODEINCOMPLETEPACKET);
1244         if(ssloppy) SSLSetSloppyMode(ctx, 0);
1245         return -1;
1246     }
1247
1248     if(ibuf->data[0] != SSL_HEADVERSION) {
1249         conn->modctx.log.update(sslLogHandle, S5_LOG_MISC, S5_LOG_ERROR,
1250                               IDS_SSL_HEADERVERSIONMISMATCH,
1251                               SSL_HEADVERSION, ibuf->data[0]);
1252         if(ssloppy) SSLSetSloppyMode(ctx, 0);
1253         return -1;
1254     }
1255     if(ibuf->data[1] != conn->state) {
1256         conn->modctx.log.update(sslLogHandle, S5_LOG_MISC, S5_LOG_ERROR,
1257                               IDS_SSL_HEADERSTATEMISMATCH,
1258                               conn->state, ibuf->data[1]);
1259         if(ssloppy) SSLSetSloppyMode(ctx, 0);
1260         return -1;
1261     }
1262
1263     ilen = ((uint8) ibuf->data[2]) << 8;
1264     ilen |= (uint8) ibuf->data[3];
1265     ilen += SSL_HEADLEN; /* we must include the header in the length because
1266                          no man is an ilen. Er, because it's the length
1267                          of the whole record, including the header. */
1268
1269 #ifdef HYPER_DEBUG
1270     if(GlobalUpdate)
1271         GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1272                     IDS_SSL_PACKETSIZE, ilen);
1273 #endif
1274
1275     if(ibuf->len < (int) (ilen)) {
1276         conn->modctx.log.update(sslLogHandle, S5_LOG_MISC, S5_LOG_ERROR,
1277                               IDS_SSL_DECODEINCOMPLETEPACKET);
1278         if(ssloppy) SSLSetSloppyMode(ctx, 0);
1279         return -1;
1280     }
1281
1282     if 0
1283     if(ibuf->len > (int) (ilen)) {
1284         conn->modctx.log.update(sslLogHandle, S5_LOG_MISC, S5_LOG_ERROR,
1285                               IDS_SSL_DECODEOVERFULLPACKET);
1286         if(ssloppy) SSLSetSloppyMode(ctx, 0);
1287         return -1;
1288     }
1289 #endif
1290
1291     SSLGetReadPendingSize(ctx, &wrtp); /* this should be zero, but seems to
1292                                         not always be, so we be safe.. */
1293
1294     if 0
1295     /* we need to choose the size of the obuf here; since SSL adds some
1296        boundary information the size of the input should be big enough.
1297        If SSL+ starts to support compression this assumption will have
1298        to change. */
1299     len = conn->readbuffer.len - conn->readbuffer.off + wrtp + ilen;
1300 #else
1301     /* OK, so we decided to change it. Now we know the record can't
1302        be larger than 16K. */
```

```
1303 /* len = conn->readbuffer.len - conn->readbuffer.off + wrtp + 16384; */
1304 /* Hmm.. the above seems to cause telnet to studder, while a fixed 32k
1305    size fixes it, so 32k it shall be... */
1306 len = 32767;
1307 #endif
1308 if(obuf->data != NULL) {
1309     if(obuf->len < (int) len) {
1310         if(GlobalUpdate)
1311             GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_DEBUG,
1312                           IDS_SSL_BUFFERTOOSMALL, obuf->len, len);
1313         obuf->len = (int) len;
1314         if(ssloppy) SSLSetSloppyMode(ctx, 0);
1315         return ENCODE_BUFFER_TOO_SMALL;
1316     }
1317 } else {
1318 #ifndef _WINDOWS
1319     obuf->data = (unsigned char *) malloc(len);
1320 #else
1321 #ifdef WIN32
1322     obuf->data = HeapAlloc(GetProcessHeap(), 0, len);
1323 #endif
1324 #endif
1325 }
1326 #if 0
1327 /* must read all the data we can.. */
1328 len = ilen + conn->readbuffer.len;
1329 #endif
1330
1331 if (conn->readbuffer.data == NULL) {
1332 /*     conn->readbuffer.data = malloc((size_t) (len - SSL_HEADLEN)); */
1333     /* Try to re-use the input buffer instead of needing to create
1334        a new one and memcpy into it. readflag lets us know we did
1335        this so we don't try to change or free the space later */
1336     conn->readbuffer.data = ibuf->data;
1337     conn->readbuffer.len = ilen;
1338     conn->readbuffer.off = SSL_HEADLEN;
1339     conn->readflag = SSL_FLOW_READ_NOOWNBUF;
1340 } else {
1341     conn->readbuffer.data = realloc(conn->readbuffer.data,
1342                                     (size_t) (len - SSL_HEADLEN));
1343     conn->readflag = 0;
1344     if(conn->readbuffer.data == NULL) {
1345         conn->modctx.log.update(sslLogHandle, S5_LOG_MISC, S5_LOG_ERROR,
1346                               IDS_SSL_REALLOCFAILED);
1347         if(ssloppy) SSLSetSloppyMode(ctx, 0);
1348         return -1;
1349     }
1350     memcpy(conn->readbuffer.data + conn->readbuffer.len,
1351            ibuf->data + SSL_HEADLEN, (size_t) (ilen - SSL_HEADLEN));
1352     conn->readbuffer.len += ilen - SSL_HEADLEN;
1353 }
1354
1355 if((err = SSLRead((void *) obuf->data, &len, ctx)) &&
1356    (err != SSLWouldBlockErr)) {
1357     conn->modctx.log.update(sslLogHandle, S5_LOG_MISC, S5_LOG_ERROR,
1358                           IDS_SSL_READERROR, err);
1359     if(ssloppy) SSLSetSloppyMode(ctx, 0);
1360     return -1;
1361 }
1362 obuf->len = (int) len;
1363 #ifdef HYPER_DEBUG
1364 if(GlobalUpdate)
1365     GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1366                 IDS_SSL_ENCODERRETURNINGBYTES,
1367                 ilen, len);
1368 if(GlobalUpdate)
1369     GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_VERBOSE,
1370                 IDS_SSL_READBUFFERGOINGOUT,
1371                 conn->readbuffer.len - conn->readbuffer.off);
1372 #ifndef AUTOSOCKS
1373     for(i = 0; i < len; i++)
```

```
1374         FPRINTF(stderr, _T("%02x "), obuf->data[i]);
1375     FPRINTF(stderr, _T("\n"));
1376 #endif
1377 #endif
1378     if(conn->readflag & SSL_FLOW_READ_NOOWNBUF)
1379         if (conn->readbuffer.off < conn->readbuffer.len) {
1380             BYTE *t;
1381
1382             if(GlobalUpdate)
1383                 GlobalUpdate(sslLogHandle, S5_LOG_MISC, S5_LOG_WARNING,
1384                               IDS_SSL_ENCODELEAVINGDATA,
1385                               conn->readbuffer.len - conn->readbuffer.off);
1386             t = malloc(conn->readbuffer.len - conn->readbuffer.off);
1387             memcpy(t, conn->readbuffer.data + conn->readbuffer.off,
1388                  conn->readbuffer.len - conn->readbuffer.off);
1389             conn->readbuffer.data = t;
1390             conn->readbuffer.len = conn->readbuffer.len - conn->readbuffer.off;
1391             conn->readbuffer.off = 0;
1392         } else {
1393             conn->readbuffer.data = NULL;
1394             conn->readbuffer.off = 0;
1395             conn->readbuffer.len = 0;
1396         }
1397     if(ssloppy) SSLSetSloppyMode(ctx, 0);
1398     return (int) ilen;
1399 }
```